Санкт-Петербургский Государственный Университет
Факультет Прикладной математики – процессов управления

Лабораторная работа по курсу «Алгоритмы и анализ сложности» на тему «Эмпирический анализ алгоритма»

Автор: Макоев Артур
331 группа ФИИТ

Алгоритм LSD поразрядной сортировки

Содержание:

История алгоритма:

Идея алгоритма поразрядной сортировки была предложена Германом Холлеритом (Herman Hollerith) в патенте 1889 года *"Art Of Compiling Statistics"*. Во время своей работы в Бюро Переписи Населения в Нью-Йорке он создал машину-табулятор, позволяющую оптимизировать обработку данных. Его изобретение выиграло среди нескольких альтернатив и было применено для обработки результатов переписи населения 1890 года. Благодаря его алгоритму, время, затраченное на обработку, сократилось до двух с половиной лет (втрое меньше, чем обработка предыдущей переписи населения).

Герман Холлерит основал собственную компанию *Tabulating Machine Company*, из которой в последствии выросла компания *International Business Machines Corporation* (IBM). В оригинальной машине, состоявшей из двух частей (tabulator и sorter), сортировка производилась начиная с наибольшего разряда. Считается, что сортировать с наименьшего разряда впервые предложил неизвестный оператор подобной машины. Первое достоверное упоминание такой технологии было в книге Роберта Фейндлера (Robert Feindler, Das Hollerith-Lochkarten-Verfahren (Berlin: Reimar Hobbing, 1929)).

При переносе алгоритма с механических устройств на компьютеры возникла проблема с выделением памяти на массивы (корзины) неизвестной длины под каждое возможное значение разряда. Эту проблему решил Гарольд Х. Сьюворд (Harold H. Seward) в 1954 году. Он предложил предварительно подсчитывать количество чисел в каждой корзине, а затем распределять их по массиву, длина которого равна длине входных данных, сохраняя значения отступов для каждой корзины.

Описание алгоритма LSD сортировки:

Алгоритм LSD поразрядной сортировки предназначен для сортировки данных, имеющих дискретные разряды. Данные сортируются по каждому разряду с помощью устойчивой сортировки, начиная с наименьшего разряда. Для сортировки по одному разряду применяется корзинная сортировка.

Вход: массив A размера n m - разрядных чисел разрядности k.

1. Создается массив B размера n и массив C размера k.
2. Для каждого разряда, начиная с наименьшего, проводится следующие операции:
    1. подсчет количества каждого из k значений в текущем разряде всех чисел, сохранение этих данных в массиве C,
    2. выделение места под корзины размеров $C[i]$ в массиве B путем подсчета смещения для хранения первых элементов со значением i в текущем разряде упорядоченного массива и сохранение этих смещений в $C[i]$,
    3. копирование элементов из A в B в места, соответствующие смещениям, записанным в C,
    4. копирование содержимого B в массив A.

3. Удаление массивов B и C.

Выход: искомый отсортированный массив хранится в A.

Псевдокод:

```
function radixSort(int[] A):
     for i = 1 to m
         for j = 0 to k - 1
             C[j] = 0
         for j = 0 to n - 1
             d = digit(A[j], i)
             C[d]++
         count = 0
         for j = 0 to k - 1
             tmp = C[j]
             C[j] = count
             count += tmp
         for j = 0 to n - 1
             d = digit(A[j], i)
             B[C[d]] = A[j]
             C[d]++
         A = B
```

Сложность LSD-сортировки

Сложность алгоритма LSD-сортировки по памяти равна $O(n + k)$: необходимо хранить два массива B и C.

Временная сложность равна $O(m \cdot n)$: необходимо проводить действия последовательно над каждым из m разрядов, где $m = const$ конечное число. Внутри итерации цикла необходимо совершить 2 обхода по массиву B длины n и 2 обхода по массиву C длины k. Над каждым элементом этих массивов производят константное число операций. При условии, что k – конечное число, сильно меньшее n, можно принять что внутри итерации цикла производят $O(n)$ операций. Итого сложность $T(n) = m \cdot O(n) =$
$= O(m \cdot n)$ операций.

Описание генератора входных данных и функции трудоемкости

В качестве меры трудоемкости я выбрал количество операций, относящихся непосредственно к алгоритму в его программной реализации на C++. Все выбранные операции работают за константное время. Я не стал выбирать время выполнения программы в качестве меры трудоемкости, потому что это потребовало бы несколько прогонов алгоритма по одним и тем же данным для уменьшения погрешности и могло приводить к разным значениям в зависимости от состояния системы. Также в алгоритме я предусмотрел возможную оптимизацию и подсчитал, сколько раз она использовалась в экспериментах.

В качестве входных данных для алгоритма LSD сортировки я выбрал 64 битные беззнаковые двоичные целые числа от 0 до $2^m - 1$, где $m = 1{,}64$. Использовал генератор псевдослучайных чисел с равномерным распределением. Провел три эксперимента:

1. $n = 10\text{^}4$ по $1.6 \cdot 10\text{^}5$ с шагом в $10\text{^}4$ и $m = 1$ по 64 по 50 различных тестов на один набор (n, m).
2. $n = 10\text{^}6$ и $m = 64$ с $10\text{^}4$ тестами.
3. $n = 10$ по 1000, $m = 64$, $10\text{^}4$ тестов.

Для задания были написаны две программы. Одна на C++, проводящая тесты с выбором диапазонов значений n, m и количеством тестов на паре значений (n, m), выводящая результаты экспериментов в файл. Вторая программа на Python для визуализации и обработке данных экспериментов.

Реализация алгоритма на C++

Полный код вместе с результатами можно посмотреть здесь:
https://github.com/zarond/AlgorythmsComplexity

Объявления некоторых типов:

```
#include <cstdint>

typedef uint64_t T;  // тип данных, используемый в эксперименте

struct testCmplxty { // тип данных, возвращаемый функцией. Сведения об одном эксперименте
	T actions = 0;
	T skips = 0;
};
```

Алгоритм сортировки без подсчета сложности:

```
// функция сортировки без подсчета количества операций
void LSD_RadixSortBinary(T* A, unsigned int n, unsigned int m) {
	// n - количество чисел
	// m - количество разрядов
	// разрядность = 2
	unsigned int* C = new unsigned int[2];
	T* B = new T[n];
	for (T i = 1; i << (63 - m); i <<= 1) {
		C[0] = C[1] = 0;
		for (unsigned int j = 0; j < n; ++j) {
			(A[j] & i) ? 0 : ++C[1]; // оптимизация - считать только количество нулей и сразу формировать отступ для корзины единиц.
		}
		if (C[1] == 0 || C[1] == n) continue; // потенциальная оптимизация в случае если нет необходимости сортировать по текущему разряду.
		for (unsigned int j = 0; j < n; ++j) {
			B[C[bool(A[j] & i)]++] = A[j];
		}
		for (unsigned int j = 0; j < n; ++j) {
			A[j] = B[j];
		}
	}
	delete[] C;
	delete[] B;
	return;
}
```

Алгоритм сортировки с подсчетом сложности:

```
// функция сортировки с подсчетом количества операций
```

```
testCmplxty LSD_RadixSortBinaryWithComplexity(T* A, unsigned int n, unsigned int m) {
	// n - количество чисел
	// m - количество разрядов
	// разрядность = 2
	T assign = 0, cmpr = 0, bitshft = 0, bitwand = 0, incr = 0; // переменные для
подсчета количества операций
	T skips = 0; // переменная для подсчета количества пропусков разряда
	//////////////////
	unsigned int* C = new unsigned int[2];
	T* B = new T[n];
	for (T i = 1; i << (64 - m); i <<= 1) {
		C[0] = C[1] = 0;
		assign += 2;
		for (unsigned int j = 0; j < n; ++j) {
			(A[j] & i) ? 0 : ++C[1] + incr++;
			++cmpr;
			++bitwand;
		}
		cmpr+=2;
		if (C[1] == 0 || C[1] == n) { ++skips; continue; }
		for (unsigned int j = 0; j < n; ++j) {
			B[C[bool(A[j] & i)]++] = A[j];
			assign++; bitwand++; incr++;
		}
		for (unsigned int j = 0; j < n; ++j) {
			A[j] = B[j];
			assign++;
		}
	}
	delete[] C;
	delete[] B;
	testCmplxty test;
	test.actions = (assign + cmpr + bitwand + incr + bitshft);   // возвращается
суммарное количество операций при сортировке
	test.skips = skips;
	// и сколько раз применился пропуск разряда
	return test;
}
```

Код генератора входных данных:

```
#include "generator.h"

// Функция для проверки, отсортирован ли массив
bool ValidateArray(T* A, unsigned int n) {
	for (int i = 0; i < n - 1; ++i)
		if (A[i] > A[i + 1]) return false;
	return true;
}

// Провести один эксперимент для данных массива Data, n количества элементов массива и m
- максимального количества разрядов
testCmplxty performtest(testCmplxty(*sorter)(T*, unsigned int, unsigned int), T* Data,
unsigned int n, unsigned int m) {
	T* A = new T[n];
	testCmplxty complexity;

	memcpy(A, Data, sizeof(T)*n);
	complexity = sorter(A, n, m);
	if (ValidateArray(A, n) == false) throw("array is not sorted");

	delete[] A;
	return complexity;
}
```

```
// Сгенерировать данные и провести repeats экспериментов на этих данных, записать
результаты в файлы file и fileskips
void begin(testCmplxty(*sorter)(T*, unsigned int, unsigned int), std::ofstream &file,
std::ofstream &fileskips, unsigned int n, int maxpower, int repeats) {
	std::random_device rd;  //Will be used to obtain a seed for the random number
engine
	std::mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd()
	std::uniform_int_distribution<T> dis(0);

	T* Data;

	Data = new T[n];

	testCmplxty complexity;

	for (int j = 0; j < repeats; ++j) {
		for (int i = 0; i < n; ++i) {
			Data[i] = dis(gen)>>(64-maxpower);
		}

		complexity = performtest(sorter, Data, n, maxpower);
		file << complexity.actions << ' ';
		fileskips << complexity.skips << ' ';
	}

	delete[] Data;
}
```

Входная функция программы:

```
#include <iostream>
#include "generator.h"
#include "LSDSort.h"

int main() {
	std::ofstream file, fileskips;
	file.open("out.txt");
	fileskips.open("outskips.txt");
	if (&file == nullptr || &fileskips == nullptr) return 1;

	unsigned int minpower = 0, maxpower = 0, minind = 0, maxind = 0, repeats = 0, numb
= 0;

	std::cin >> minpower >> maxpower >> minind >> maxind >> repeats >> numb; // ввод
параметров для эксперимента из консоли

	if (minpower == 0 || minpower > maxpower || minind == 0 || minind > maxind ||
repeats == 0 || numb == 0) return 2;

	for (int i = minpower; i <= maxpower; ++i) {
		for (int j = minind; j <= maxind; ++j) {
			T n = j * numb;
			file << n << ' ' << i << ' ';
			fileskips << n << ' ' << i << ' ';
			begin(LSD_RadixSortBinaryWithComplexity, file, fileskips, n, i,
repeats);
			file << std::endl;
			fileskips << std::endl;
		}
	}

	file.close();
```

```
        fileskips.close();
        system("pause");
        //system("python graph.py");
}
```

Код на Python

Анализ и построение графиков:

“graph1.py”

```
import numpy as np
import matplotlib.pyplot as plt
import scipy, scipy.stats

print("started plotting")

file = open("experiments/uniformdistr.txt")
file1 = open("experiments/ConstN.txt")
file2 = open("experiments/small1.txt")
#fileskip = open("outskips.txt")

X, Y, Z = [],[],[]
for line in file:
  l = [int(s) for s in line.split()]
  Y.append(l[0])
  Z.append(l[1])
  X.append(l[2:])

X1, Y1 = [],[]
for line in file1:
  l = [int(s) for s in line.split()]
  Y1.append(l[0])
  X1.append(l[2:])

X = np.array(X)
Y = np.array(Y)
Z = np.array(Z)
X1 = np.array(X1)
Y1 = np.array(Y1)
fig,(ax,ax1) = plt.subplots(ncols=2, constrained_layout=False)
for i in range(len(X)):
    ax.plot(Z[i]*np.ones(len(X[i])),X[i],marker = '.')

ax.grid()
ax.set(xlabel='x: number of digits I', ylabel='y: number of operations')

for i in range(len(X)//64):
    ax.text(65,X[-1-i][0],str(str(16-i)+'*10^4'))

for i in range(16):
    line = np.polyfit(np.repeat(Z[i::16],50),np.ndarray.flatten(X[i::16][:]),1)
    ax.plot([0,65],[line[1],line[0]*65+line[1]],color = 'black', alpha = 0.5, linestyle = '--
')

for i in range(len(X)):
    ax1.plot(Y[i]*np.ones(len(X[i])),X[i],marker = '.')

ax1.grid()
ax1.set(xlabel='x: array length N', ylabel='y: number of operations')

for i in range(64):
    ax1.text(163000,X[(i*16+15)][0],str(i+1))

for i in range(64):
    line = np.polyfit(np.repeat(Y[0:16],50),np.ndarray.flatten(X[i*16:(i*16+16)][:]),1)
    ax1.plot([0,160000],[line[1],line[0]*160000+line[1]],color = 'black', alpha = 0.5,
linestyle = '--')
```

```
fig.suptitle('number of binary digits I = 1:64, array length N = 10^4:16*10^4, number of
repeated tests with same array length and digits = 50', fontsize=16)
plt.show()

fig,ax = plt.subplots()
A = np.sort(X1[0])
pmf = scipy.stats.binom.pmf(A-64*(6*10**6+4),64000000,0.5)
ax.plot(A,pmf)
ax.hist(A,100,normed=True)
ax.set(xlabel='x: number of operations N', ylabel='distribution',
    title='distribution of number of operations. number of digits I = 64, array length N =
10^6, number of tests M = 10^4')
plt.show()



fig,ax = plt.subplots()

X, Y, Z = [],[],[]
for line in file2:
  l = [int(s) for s in line.split()]
  Y.append(l[0])
  Z.append(l[1])
  X.append(l[2:])

for i in range(len(X)):
    ax.plot(Y[i]*np.ones(len(X[i])),X[i],marker = '.')

ax.grid()
ax.set(xlabel='x: number of elements n', ylabel='y: number of operations')
fig.suptitle('number of operations on small sets. n = 10 to 1000, m = 64, 10000 repeats')
X = np.array(X)
line = np.polyfit(np.repeat(Y[:],10000),np.ndarray.flatten(X[:][:]),1)
ax.plot([0,1000],[line[1],line[0]*1000+line[1]],color = 'black', alpha = 0.5, linestyle = '--
')

plt.show()

fig,ax = plt.subplots()
A = np.sort(X[-1])
pmf = scipy.stats.binom.pmf(A-64*(6*1000+4),64000,0.5)
ax.plot(A,pmf)
ax.hist(A,100,normed=True)
ax.set(xlabel='x: number of operations N', ylabel='y: distribution')
fig.suptitle('distribution n = 1000, m = 64, 10000 repeats')

plt.show()

file.close()
```

“graph2.py”

```
import numpy as np
import matplotlib.pyplot as plt
import scipy, scipy.stats

print("started plotting")

file = open("experiments/uniformdistr.txt")
file1 = open("experiments/double11.txt")
file2 = open("experiments/double12.txt")
#fileskip = open("outskips.txt")

X1, X2, Z = [],[],[]

for line in file1:
  l = [int(s) for s in line.split()]
```

```
  X1.append(l[2:])
  Z.append(l[0])

for line in file2:
  l = [int(s) for s in line.split()]
  X2.append(l[2:])

X1 = np.array(X1)
X2 = np.array(X2)
X1 = np.mean(X1,axis = 1)
X2 = np.mean(X2,axis = 1)
Z = np.array(Z)

Y = X2/X1

fig,ax = plt.subplots()
ax.grid()
ax.set(xlabel='x: N - length of array, from 10^3 to 10^6 with 10^3 step. 5 repeats for each
N', ylabel='y: T(2n)/T(n)')
fig.suptitle('T(2n)/T(n)')
ax.plot(Z,Y)
plt.show()
```

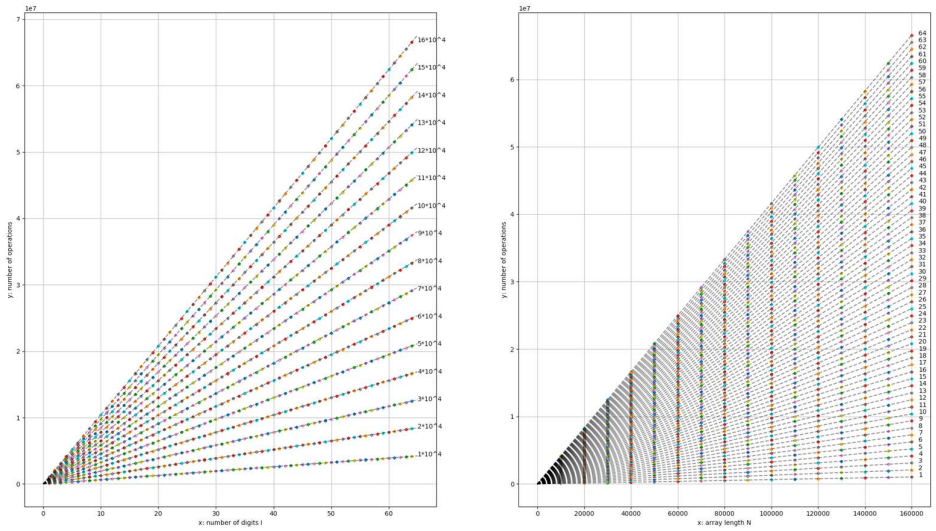
Результаты:

В алгоритме LSD сортировки мало возможностей по оптимизации. Я применил две оптимизации: считал только “0” и рассматривал возможность пропустить копирование данных, если в текущем разряде у всех чисел одинаковое значение. Однако за все проведенные эксперименты на массивах длиной больше 10 элементов пропуск разряда ни разу не сработал. Что ожидаемо, потому что вероятность того, что у всех чисел будет одинаковое значение разряда, экспоненциально стремится к нулю при росте числа элементов. Первая оптимизация, однако, дала некоторый результат, но уменьшение сложности от ее применения было незначительным.

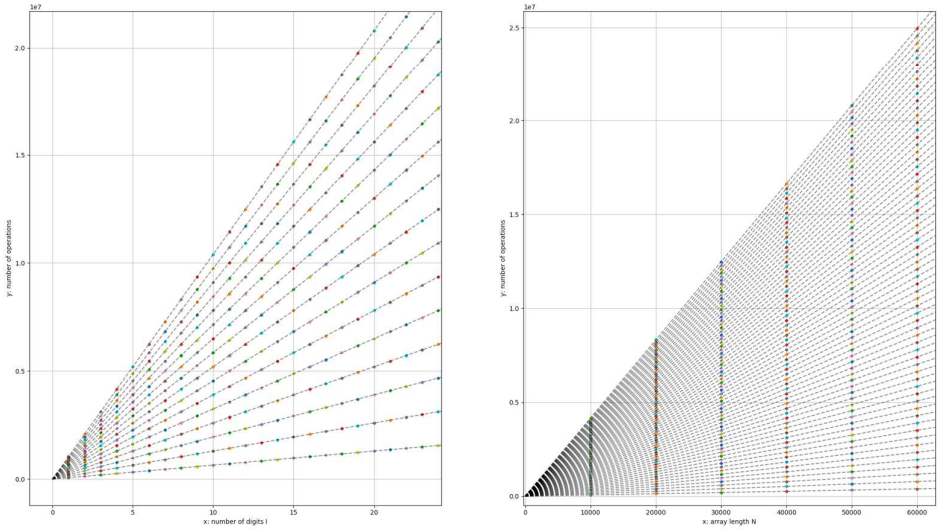
Программа выдает результаты в файлы в виде файлов *out.txt* и *outskips.txt* в формате: $n, m,$ результаты вычисления. На каждой строке по новой паре значений (n, m).

На следующих изображениях представлены результаты выполнения python скрипта *graph1.py*, который визуализирует данные трех экспериментов: первый с диапазоном значений для n и m, второй с константными параметрами $n = 10\text{^}6$, $m = 64$ и с $10\text{^}4$ повторениями эксперимента, третий с малыми значениями n. По результатам первого построены графики, по результатам второго – гистограмма частот, по результатам третьего – график и гистограмма частот. К данным первого эксперимента была построена линейная аппроксимация методом наименьших квадратов, хорошо проходящая через вычисленные значения. Было подтверждено предположение, что функция трудоемкости линейна по обоим параметрам.

number of binary digits l = 1:64, array length N = 10^4:16*10^4, number of repeated tests with same array length and digits = 50

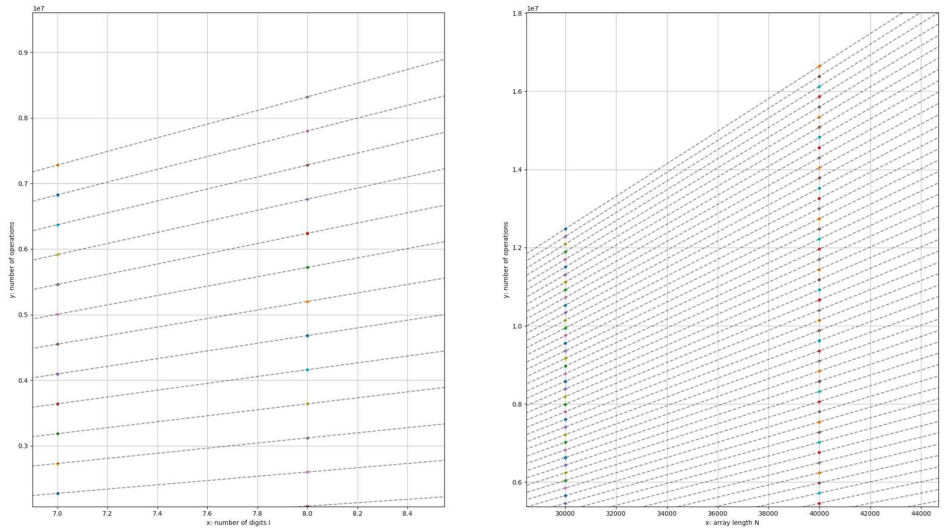


number of binary digits l = 1:64, array length N = 10^4:16*10^4, number of repeated tests with same array length and digits = 50



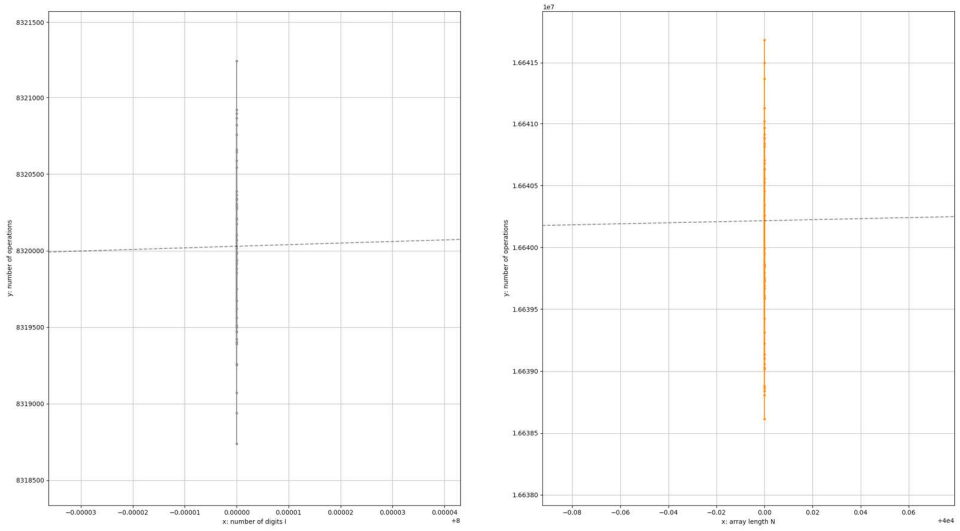
zoom rect

number of binary digits I = 1:64, array length N = 10^4:16*10^4, number of repeated tests with same array length and digits = 50



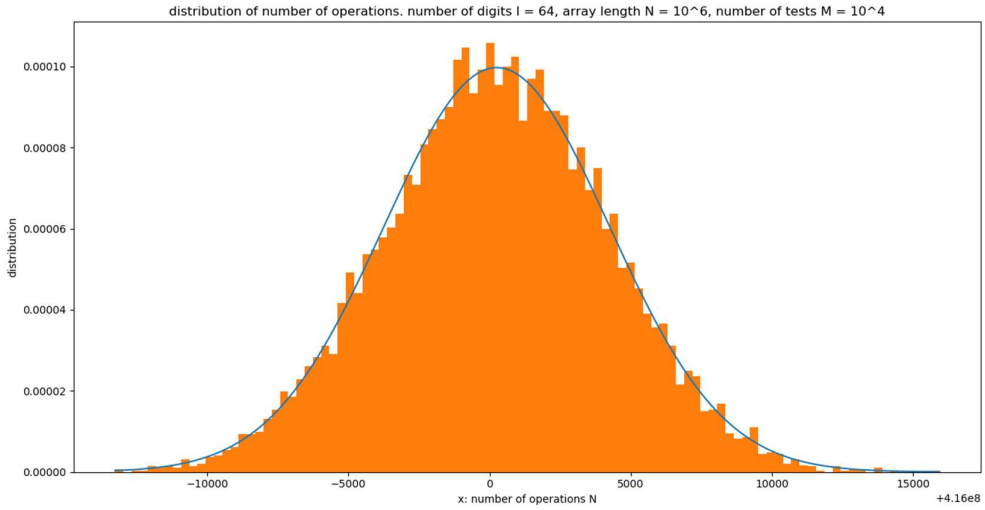
zoom rect

number of binary digits I = 1:64, array length N = 10^4:16*10^4, number of repeated tests with same array length and digits = 50



zoom rect

Для второго эксперимента была получена гистограмма частот.



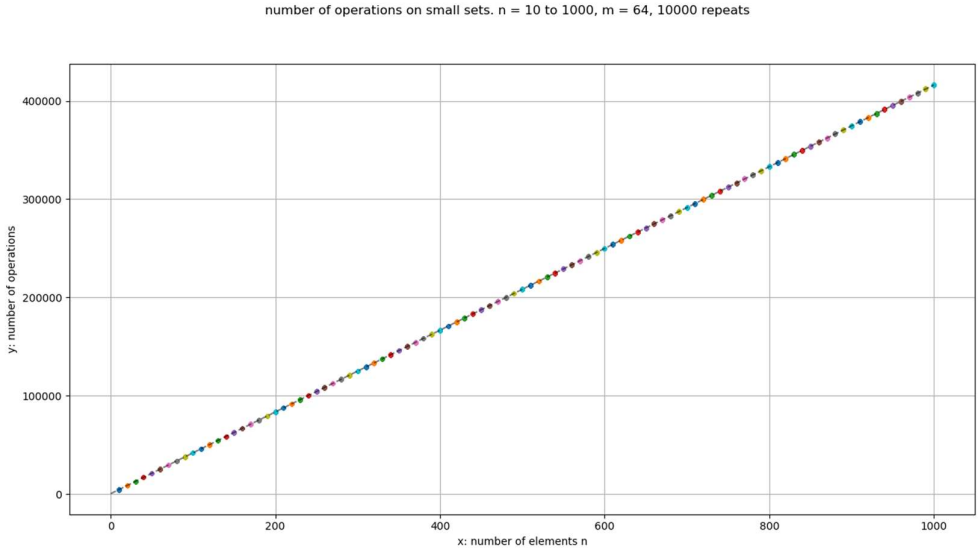
Наблюдаемая величина обладает полученным распределением, потому что в алгоритме считается только количество нулей в бинарном представлении числа. Операция выполняется, только если в текущем разряде числа стоит нуль. То есть, случайная величина есть суммарное количество нулей в разрядах всех чисел массива. Для равномерно распределенной случайной величины (значения элемента массива), вероятности иметь в любом разряде нуль равны 0.5, потому что каждый разряд разделяет числовую ось на два равных по мощности множества: в первом текущий разряд равен 0, во втором равен 1. Таким образом, искомая случайная величина – количество равновероятных успехов (выпадения 0) среди $64 \cdot n$ испытаний. То есть искомое распределение – биномиальное $B(n,p)$: $P(k) = \binom{n}{k} \cdot p^k \cdot q^{n-k}$. В данном случае параметры распределения $p = 0.5$ и $n = 10\text{^}6$. Математическое ожидание $M = n \cdot p = 0.5 \cdot 64 \cdot 10\text{^}6 = 3.2 \cdot 10\text{^}7$, дисперсия $D = n \cdot p \cdot q = 0.25 \cdot 64 \cdot 10\text{^}6 = 1.6 \cdot 10\text{^}7$, стандартное отклонение $\sigma = \sqrt{1.6 * 10\text{^}7} = 4000$. На графике синей линией представлено биномиальное распределение с такими параметрами, смещенное на $64 \cdot (6 \cdot 10\text{^}6 + 4)$ – теоретическое количество операций, которые выполняются на каждом эксперименте.

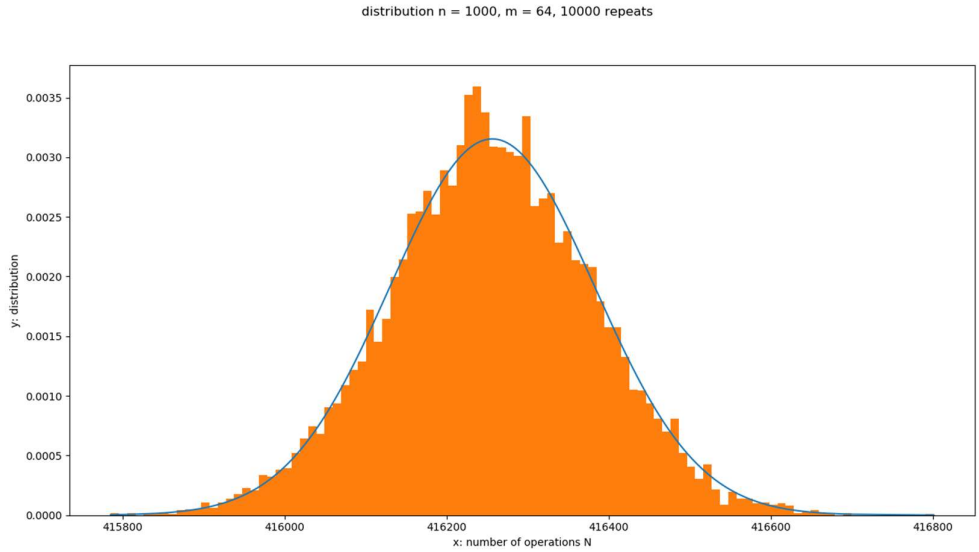
Рассмотрим, сколько процентов составляет максимальное отклонение данных от математического ожидания по отношению к математическому ожиданию.

$$15 \cdot 10\text{^}3/(4.16 \cdot 10\text{^}8) = 3.5 \cdot 10\text{^}(-3)\ \%$$

Таким образом, уменьшение числа операций из-за оптимизации очень незначительное, и такие оптимизации при реализации алгоритма можно опускать.

Для третьего эксперимента была получена линейная аппроксимация и гистограмма частот





Если принимать во внимание то, что количество операций подчиняется биномиальному распределению со сдвигом, с математическим ожиданием $M = 0.5 \cdot m \cdot n$, то сложность в среднем случае будет $T(n) = m \cdot (6 \cdot n + 4) + 0.5 \cdot m \cdot n = m \cdot (6.5 \cdot n + 4) = O(m \cdot n)$. В лучшем случае, если сработали все пропуски разрядов, все равно необходимо m раз пройти по массиву, то есть $T = O(m \cdot n)$.

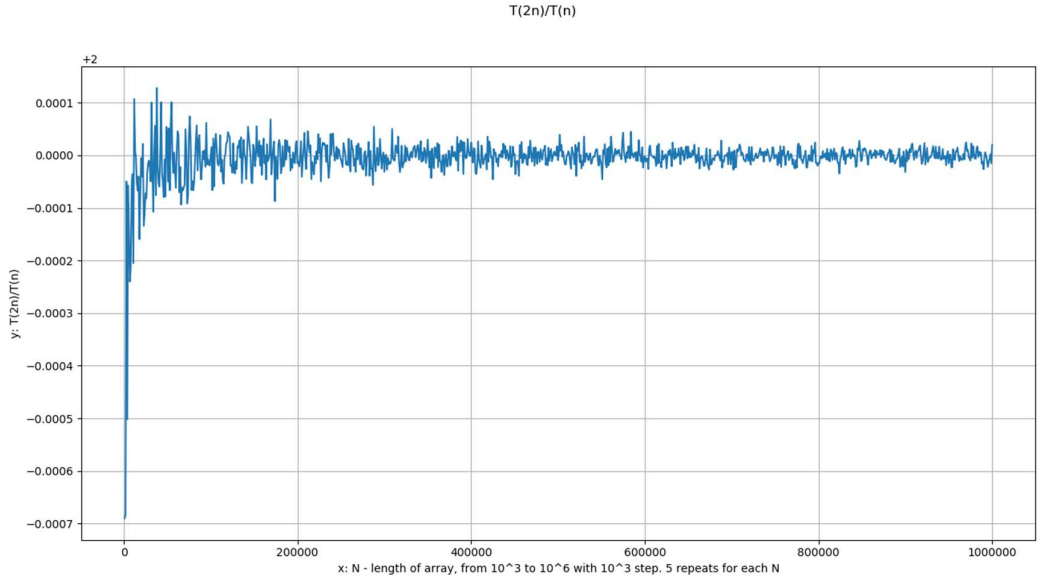
Таким образом, сложность алгоритма бинарной поразрядной LSD сортировки равна $O(m \cdot n)$ в худшем, лучшем и среднем случае.

Исследование функции трудоемкости при удвоении входных данных.

Рассмотрим поведение функции $T(2 \cdot n)/T(n)$ при больших n.

Если функция трудоемкости принадлежит классу $O(n^k)$, то при удвоении входных данных функция $T(2 \cdot n)/T(n)$ асимптотически стремится к значению 2^k.

В нашем случае это отношение должно стремиться к 2.



На картинке – результат работы программы *graph2.py*, визуализирующей данный вычислительный эксперимент (файлы double11 и double12). Как видно из графика, значение функции действительно приближается к y = 2 при увеличении N.

Характеристики вычислительной системы:

Процессор Intel Core i5-6300HQ CPU @ 2.30GHz

Оперативная память 32 Гб

Тип системы: 64-разрядная операционная система Windows 10, процессор x64

Источники:

1. **http://patft.uspto.gov/netacgi/nph-Parser?patentnumber=395781**
   http://patft.uspto.gov/netacgi/nph-Parser?patentnumber=395782
   http://patft.uspto.gov/netacgi/nph-Parser?patentnumber=395783
   - оригинальные патенты Германа Холлерита.
2. http://www.columbia.edu/cu/computinghistory/hollerith.html
   http://www.columbia.edu/cu/computinghistory/census-tabulator.html
   - сайт Колумбийского университета
3. Искусство программирования. Том 3. Сортировка и поиск. Глава 5.5 стр. 416
   (The Art of Computer Programming: Volume 3: Sorting and Searching)
   Автор: Дональд Кнут (Donald E. Knuth)
4. https://www.cs.princeton.edu/~rs/AlgsDS07/18RadixSort.pdf - сайт Принстонского университета США
5. https://neerc.ifmo.ru/wiki/index.php?title=%D0%A6%D0%B8%D1%84%D1%80%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0 – вики университета ИТМО
6. https://medium.com/basecs/getting-to-the-root-of-sorting-with-radix-sort-f8e9240d4224